unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(12)

AD-A188 706

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 87-11-07

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Northwest Laboratory for Integrated Systems
University of Washington
Semiannual Technical Report No. 5

5. TYPE OF REPORT & PERIOD COVERED
Technical, interim

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Northwest Laboratory for Integrated Systems

8. CONTRACT OR GRANT NUMBER(s)
MDA903-85-K-0072
ARPA-4563/2 Code 5D30

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Northwest Laboratory for Integrated Systems
University of Washington, Computer Science
Seattle, WA 98195 FR-35

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
DARPA-ISTO
1400 Wilson Boulevard
Arlington, VA 22209

12. REPORT DATE
November, 1987

13. NUMBER OF PAGES
38

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
ONR
University of Washington
315 University District Building
1107 NE 45th St., JD-16, Seattle, WA 98195

15. SECURITY CLASS. (of this report)
unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

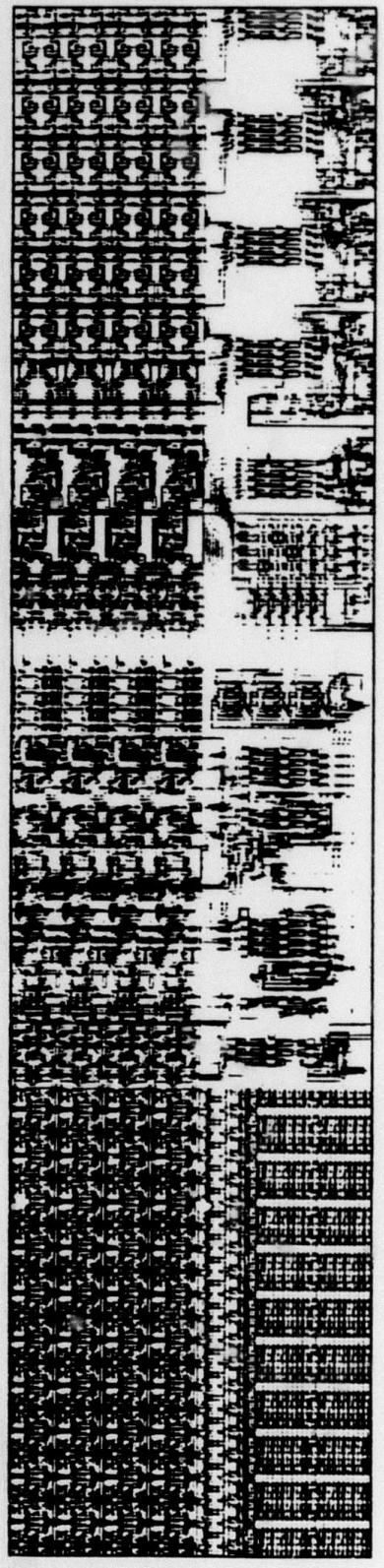
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
VLSI Design Generators, VLSI architectures, and CAD, Network, C, Apex, CMOS, NC, Netlist, Circuit Parallelism, Declarative Descriptions, Bezier curves

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This document reports on the research activities of the Northwest Laboratory for Integrated Systems, formerly the UW/NW VLSI Consortium, for the period of April 7, 1987 to November 16, 1987, under the sponsorship of the Defense Advanced Research Projects Agency, under contract number MDA903-85-K-0072, program code number 5D30.

(Keywords:)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



NORTHWEST LIS

(LABORATORY FOR INTEGRATED SYSTEMS)

Semiannual Technical Report No. 5
"VLSI Design Generators"
University of Washington

November 16, 1987
TR # 87-11-07

Reporting Period: 7 April–16 November 1987
Principal Investigator: Lawrence Snyder

Sponsored by:
Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 4563/2
Issued by Defense Supply Service-Washington
Under Contract #MDA903-85-K-0072
(Program Code Number: 5D30)

Contents

1 Overview of Activities

1.1 Design Generators

A full custom VLSI chip design is typically composed of a few interesting and unique circuits and numerous commonly-used circuits such as RAMs, ROMs, PLAs, MUXes, decoders and the like. A central research question facing architects of CAD systems is: How can one easily capture the expertise that goes into the construction of these commonly-used circuits so that they may be easily modified and re-used in one chip after another?

Our answer to this question has been to develop a methodology for constructing design generators – programs that produce entire families of circuit designs. Existing design generator systems are limited by either a rigid implementation strategy or by the inability to support multiple descriptions of the circuit. Our strategy has been to develop an open-ended system that does not restrict the implementation strategy, and supports the capture of several descriptions of a family of circuits – layout, schematic and simulation model.

1.2 Multilevel Simulation

One of the major problems in the design of electronic circuits is how to simulate such systems at a variety of levels of abstraction. Ideally the designer would like an environment in which he or she could specify and simulate an entire architecture at a very high level of abstraction, while some particular part of the whole is simulated at a lower level, e.g. at the gate, switch or even analog level. In other words the high level architectural model serves as a context for simulating the detailed circuit level model.

Our attempt to construct such a multilevel simulator has led to the development of Network C (NC), a language-based simulation environment. Network C supports discrete event scheduling for system modeling and a general nonlinear equation solver for analog modeling. We are currently evaluating several proposals for refinement of both the specification language as well as the run-time support.

1.3 Architectural Experiments

Several ongoing architectural experiments provide testbeds for the generators project as well as our multilevel simulation work. In 1985 students in an advanced VLSI class designed a 32 bit microprocessor known as the Quarter Horse, which turned into a case study of choice complexity. Within the past year, work has progressed on two designs for a chip that generates curves and surfaces from sets of control points. The two designs explore fundamentally contrasting approaches to the problem – that of a high performance, fixed arrangement of multiple processing elements vs a single programmable processing element.

2 Future Plans

In the proposed contract "VLSI Architectures and CAD" we describe two research challenges: Exploit VLSI in high performance computer architectures and empower VLSI CAD with greater sophistication for computer architecture design. Our strategy is to focus on the interface between computer architecture and VLSI CAD, so that the the qualities of each are present in the results of the other. From this investigation should come architectures leveraging rather than fighting the characteristics of VLSI, and VLSI CAD tools supporting rather than impeding architecture design. Two vehicles have been chosen for this study. On the architecture side we will design an innovative processor element architecture applicable to the next generations of parallel computers. This PE design will leverage custom VLSI and exploit the special features of parallel processors to deliver dramatically better performance. On the VLSI CAD side our intent is to provide powerful and versatile design tools for complex systems containing some custom chips and a variety of other components. We are already examining a multilevel simulation system, functional language specification of circuits, and a novel fast-search approach to leaf cell design.

3 Current Activities

3.1 Progress In Design Generators

(Larry Snyder, Jean-Loup Baer, Larry McMurchie, Wayne Winder, Rudolf Nottrott)

The generators project has reached a milestone with the completion of programs that generate layout, schematic diagram and network decriptions (see Appendix A). The input to these programs is 1) a notation that describes the geometric and network structures of an entire family of circuits and 2) a list of parameters that specify the particular instance circuit desired. We believe the notation provides a compact hierarchical specification of circuit structure and that it is a convenient means of capturing circuit expertise for others to utilize.

One of the most interesting aspects of developing this notation has been the analysis of specifications for circuit geometry and network structure. In particular we have tried to find the elements common to both representations so that they can be exploited in the notation. Obviously a hierarchy of design objects is one such common element. Also, we found how useful network information could be in constructing layouts, as it could provide the specific information about what signals are to be connected when layout objects are being positioned. By including such information layout generation could be freed from the requirement of interlocking cells.

We have tested the notation by implementing a number of decoders as well as a Baugh-Wooley multiplier, and are beginning to apply it to coded circuits such as PLAs and ROMs as well. Furthermore, we are looking at a number of extensions to the notation, including the addition of network information to the layout assembly.

3.2 Simulation with NETWORK C

(Bill Beckett)

The Network C (NC) simulation system has been used extensively in the development and refinement of the APEXII design (Section 3). Initially a very high level model of the APEXII algorithm was implemented in NC. Because NC is based on C language models, we were able to easily interface a library of C graphics routines to the chip model and display the curves being "generated" by the chip model. Thus a "proof of principle" experiment could be performed on the algorithm with a minimum of effort. We then began specifying the architecture of the chip. We did a block-level specification in which all the major blocks (RAMs, ROM, processor elements, datapaths etc.) and their functions were specified. Again we could simulate this block-level specification with the same graphical interface as before. In the following months we refined the architecture, again simulating the refinements by performing the curve drawing experiments.

At the same time as the architecture was being refined, one of us expanded the level of detail of the original algorithm and looked at the accuracy of the curves as a function of word size, curve type and degree. Again NC was used as the simulation framework. The results of these experiments resulted in the determination of a minimum word size to be employed in the processing element. This result was then implemented in the architectural model.

While these two efforts were proceeding, another member of the design team developed a RAM, the basic design of which was simulated using the MOS modeling capability of NC.

At this time the architectural model of APEXII has been completed. The problem we are now concerned with is ensuring the block layouts we have constructed are identical in function to the corresponding models in the NC simulation. With the smaller blocks we are able to replace the model with the extracted circuit modeled at the MOS level, while the remainder of the chip is modeled at the functional level. Several blocks are simply too large for this scheme. What is needed is a more approximate switch-level modeling of these larger blocks. The procedure we have adopted is to interface the switch level simulator RNL to NC through a fork. Again, because NC is based on C language models, implementing a fork within a model was straightforward. In the near future we intend to implement a switch level algorithm in NC; while that is in progress the RNL forking scheme provides both a temporary solution to the problem as well as a prototype of the final implementation of switch-level simulation in Network C.

The use of both the high level functional capability and the MOS modeling of NC has provided its developer with considerable feedback. Numerous modifications have been made to accommodate its use on a chip design the size of APEXII. A new user's guide which reflects much of this feedback was written and is available. We expect to be able to distribute Network C to the community during the first half of 1988.

3.3 APEX: An Architecture For Drawing Parametric Curves and Surfaces

(Carl Ebeling, Tony DeRose, Larry McMurchie, Bill Barnard, Bill Yost)

Two chips are currently being designed for the purpose of drawing curves and surfaces from sets of control points (see Appendix B). APEX I employs multiple processing elements in a triangular data-flow architecture. APEX II performs the same computation in a more flexible way that allows the generation of higher degree curves at the cost of lower performance.

In the design of APEXII a detailed architectural model was constructed and simulated with the simulation system Network C (see section above). APEXII employs single and dual ported RAMs, a ROM controller, two multipliers, two adders, a subtractor, a counter and numerous datapaths of random logic. Layout generators for some of these modules were already available. For other parts the generators had to be written.

Currently our work focuses on two areas: the first task is to verify that the layouts have the behavior of the corresponding functional models in the Network C simulation; the second task is the placement and routing of all the modules on the available silicon.

Verification of the layouts in the context of the architectural model for the chip has provided us with useful feedback on NC. The scheme we ultimately adopted is to fork an RNL process from within NC. Stimuli from surrounding blocks are applied through a UNIX pipe to the extracted circuit being simulated within RNL. This scheme has been successful in isolating clocking problems as well as labeling errors in the arithmetic blocks. Currently, layouts for all of the arithmetic blocks have been verified.

The placement and routing methodology that we started with was to manually place all the generator-created blocks and route them with MAGIC. A special utility for generating the netlist from the NC simulation model was written. The routing task, however, was simply too ambitious for MAGIC to handle all at once. With nearly 50 separate modules and over 1000 nets, MAGIC was never able to complete the route. We are currently pursuing several alternatives simultaneously. Some of the blocks are closely related and can be combined into a single datapath. The pitchmatching features of the generators allow layouts for such blocks to be generated so that they align in a datapath fashion. Another approach has been to hand route some of the more ubiquitous buses. Finally, we are dividing the remaining netlist into several parts and routing each one individually. We are confident that the combination of these efforts will result in a routed chip. Alternative solutions we have proposed and discarded have included: moving to 1.25 micron technology (from 2.0 microns), decreasing the amount of built-in testing, reducing the functionality, improving the MAGIC routing code and more hand routing.

3.4 Drawing Netlist Programs

(Carl Ebeling, Zhanbing Wu)

One of our graduate students, Zhanbing Wu, worked last summer importing the drawing program xdp (originally developed at CMU by Dario Giuse) into our environment as a means of drawing schematics. We have defined conventions for writing netlist programs as hierarchical circuit draw-

ings instead of text. These drawings allow the designer to represent arbitrary netlist programs graphically. Arbitrary Lisp expressions can be used within the drawing to specify repetitive and parameterized circuits; however, these expressions can themselves contain circuit drawings. When complete, drawings are translated by a drawing analyzer into a netlist language that can be compiled into SIM format files for simulation or for layout validation. We are now redefining the intermediate netlist language so that it will support functional circuit specifications in the style of Network C or CSIM (University of Colorado). We also plan to extend the netlist language with a better mechanism for defining signals with structure.

3.5 Investigations Into Circuit Parallelism

(Larry Snyder, Mary Bailey)

We have been investigating the question: How much parallel "activity" is there on CMOS VLSI Chips? Most researchers have assumed that large chips have many transistors firing simultaneously, but because no one can measure the activity, no one can be sure. The first problem, then, is how to determine the amount of switching on a chip. Simulation is the obvious answer, but the matter is more complicated. Transistor switching is a continuous, analog activity, but parallelism, as the term is generally used in computer science, seems to be more digital, based on the concept of a "step". Also, to further complicate the use of simulation for determining on-chip parallelism, the detailed SPICE simulations that engineers generally trust are computationally infeasible for chips with more than a few hundred devices.

We have developed a methodology, using Terman's linear-level simulator RNL, for empirically determining the amount of parallelism on a CMOS VLSI chip. We also have a "calibration" study showing to what extent RNL can serve as a surrogate for SPICE, and a study of the impact that the simulation step size has on parallelism. We applied this methodology to six CMOS chips developed at the UW. With the exception of a 16-bit shift register (which was included to exhibit substantial switching), the number of transistors switching in a 0.1 nanosecond timestep was around 5. Considering that two of the chips contained over 20,000 transistors, this is a remarkably small amount of parallelism. We also investigated how the choice of inputs and circuits size affect parallelism, and found that previous efforts which extrapolate parallelism for large circuits by measuring the parallelism in a small instance may be faulty.

If our results are correct, and CMOS VLSI chips are not very parallel, this has several implications. First, we should try and discover why this is true and use this information to discover how to increase the parallelism of circuits. We may want to use this information to investigate alternative architectures for VLSI implementation. Second, if chips are not parallel, the conventionally accepted technique of speeding up simulation through partitioning the circuit onto different processors may be doomed.

3.6 Layouts from Functional Language Specifications

(Martine Schlag, Simon Kahan)

Our work on the use of a high level functional language for the specification of integrated circuits continues and has recently focused on the problem of mapping the planar topology of a circuit to a

layout. Our previous method relied on one-dimensional compaction, assembling the circuit based on the specification by packing sequential constructs vertically and parallel constructs horizontally. The result was an efficient mapping from a functional language specification of a circuit to an abstract layout of a circuit or an actual layout if the technology and leaf cells were provided as well. Although many circuits and their layouts can be described in this manner, this method restricts the flow of signals to the vertical dimension; there was no provision for a signal to flow horizontally from one primitive operator to another.

This restriction has been removed by first adapting the compactor to handle these signals and then providing the compactor with the ability to compact in the other dimension by rotating the layout by 90 degrees. These modifications/enhancements of the current compactor were made by a graduate student, Simon Kahan, during the summer of 1987. As a result of Simon's work, we are no longer restricted to assembling sequential constructs vertically. It permits us to apply different packing strategies to sub-circuits of the design. The challenge is now to exploit the information inherent in the assembly history provided by the functional language specification to determine the compaction strategy.

3.7 A Model for Architectural Comparisons

(Larry Snyder, Sam Ho)

Recently, architectures for sequential computers have become a topic of much discussion and controversy. At the center of this storm is the Reduced Instruction Set Computer, or RISC, first described at Berkeley in 1980. While the merits of the RISC architecture cannot be ignored, its opponents have tried to do just that, while its proponents have expanded and frequently exaggerated them. This state of affairs has persisted to this day.

The problem with these arguments was that their proponents were not speaking of the same things. Each side quite naturally chose examples that most supported his own view. On top of that, the RISC I chip from Berkeley contained an essentially unrelated piece of hardware, that of multiple overlapping register sets. The early papers on RISC often combined the effects of the register set and the instruction set with little regard for their relationship, which was tenuous, at best. When the RISC I chip turned out to have an error that caused it to run extremely slowly, it provided no vindication for the proponents of the CISC, since the problem had nothing to do with the complexity of the instruction set.

In our analysis we start with a computer, defined by its functional units, such as ALU, shifter, and registers, and its control, microcode or hardwired controls. We choose a calculation, such as a matrix product or a text formatter, and decompose it into basic actions, which are arithmetic operations and their relatives. To actually implement this calculation, we will need to generate some necessary overhead actions, such as fetches and decodes. Finally, the functional units determine the cost in time units called cycles of each action. The total cost of the calculation is the sum of the number of cycles needed. The lower this number is, the faster the computer operates.

This model is an attempt to provide a common quantitative basis for a discussion of reduced vs complex instruction sets and other architectural questions. Given a set of parameters and examples, it provides a numerical result with which to compare that resulting from other such parameters. This dependence on the example cannot be ignored, and reflects the truth that the performance of

any system depends greatly on what it is being used for, as compared to what it was designed for.

3.8 NW LIS VLSI Tools RELEASE 3.1

(Warren Jessop)

Since early this year we have distributed Release 3.1 of our design toolset. Included in this release are all of the Berkeley '86 tools, in particular MAGIC as well as NETLIST/PRESIM/RNL from MIT and a design rule checker from Carnegie-Mellon. Also included is our home-grown procedural layout package CFL. Many of the generators that we have written with CFL during the past several years have also been included in 3.1.

Currently this release has been distributed to 134 sites, with 61 additional site licenses being processed.

4 Recent Publications and Reports

1) Near-Optimal Speedup of Graphics Algorithms using Multiguage Parallel Computers, L. Snyder, T. DeRose and C. Yang, Proceedings of the International Conference on Parallel Processing (to appear) 1987.

2) Practical Algorithms for Image Component Labeling on SIMD Mesh Connected Computers, R. Cypher, J. Sanz and L. Snyder, Proceedings of the International Conference on Parallel Processing (to appear) 1987.

3) The Hough Transform Has O(N) Complexity on SIMD N x N Mesh Array Architectures, R. Cypher, J. Sanz and L. Snyder, Technical Report 87-07-01, University of Washington, 1987.

4) The Plan : Topology of Functional Programs, M. Schlag, The Third Functional Programming Languages and Computer Architecture Conference, Portland, Oregon, September, 1987.

5) An Investigation of Multiguage Architectures, C. Yang, Ph. D. Thesis, Technical Report 87-10-05, University of Washington, 1987.

6) Hercules: A Power Analyzer for MOS VLSI Circuits, A. Tyagi, Proceedings of the 1987 IEEE International Conference on Computer-Aided Design, pp. 530-533, 1987.

any system depends greatly on what it is being used for, as compared to what it was designed for.

3.8 NW LIS VLSI Tools RELEASE 3.1

(Warren Jessop)

Since early this year we have distributed Release 3.1 of our design toolset. Included in this release are all of the Berkeley '86 tools, in particular MAGIC as well as NETLIST/PRESIM/RNL from MIT and a design rule checker from Carnegie-Mellon. Also included is our home-grown procedural layout package CFL. Many of the generators that we have written with CFL during the past several years have also been included in 3.1.

Currently this release has been distributed to 134 sites, with 61 additional site licenses being processed.

4 Recent Publications and Reports

1) Near-Optimal Speedup of Graphics Algorithms using Multiguage Parallel Computers, L. Snyder, T. DeRose and C. Yang, Proceedings of the International Conference on Parallel Processing (to appear) 1987.

2) Practical Algorithms for Image Component Labeling on *SIMD Mesh Connected Computers*, R. Cypher, J. Sanz and L. Snyder, Proceedings of the International Conference on Parallel Processing (to appear) 1987.

3) The Hough Transform Has O(N) Complexity on SIMD N x N Mesh Array Architectures, R. Cypher, J. Sanz and L. Snyder, Technical Report 87-07-01, University of Washington, 1987.

4) The Planar Topology of Functional Programs, M. Schlag, The Third Functional Programming Languages and Computer Architecture Conference, Portland, Oregon, September, 1987.

5) An Investigation of Multiguage Architectures, C. Yang, Ph. D. Thesis, Technical Report 87-10-05, University of Washington, 1987.

6) Hercules: A Power Analyzer for MOS VLSI Circuits, A. Tyagi, Proceedings of the 1987 IEEE International Conference on Computer-Aided Design, pp. 530-533, 1987.

APPENDIX A

A NOTATION FOR DESCRIBING MULTIPLE VIEWS OF VLSI CIRCUITS

Jean-Loup Baer, Meei-Chiueh Liem, Larry McMurchie,
Rudolf Nottrott, Lawrence Snyder, Wayne Winder
NW Laboratory for Integrated Systems
Department of Computer Science
University of Washington
Seattle, WA 98195

ABSTRACT

A declarative hierarchical notation is introduced that allows the parametric representation of entire families of VLSI circuits. Layout, schematic diagrams and network structure are all accommodated by the notation in a way that emphasizes common elements. The notation is the basis of a structured environment for developing design generators as well as capturing design expertise.

1 Introduction

The application of full custom integrated circuit design to architectural problems requires a multitude of CAD techniques. Not only does the designer have to specify and simulate networks of components, but he also has to deal with the physical layout of those components. It seems apparent that the sheer volume of circuitry required for even the simplest architectures mandates an approach that utilizes previously captured circuit designs. One method of capturing expertise about entire families of circuit design is through the use of design generators.

We define a design generator as a program that produces instances from a family of circuit designs. The input is a problem-specific set of parameters; the most common output is the layout of the mask layers. Several systems have been developed for the construction of layout generators (e.g.[Mayo 83], [Bamji 85]).

Although the layout is the final means of specifying a circuit, it is simply too detailed and technology-dependent to efficiently capture design expertise. Other more abstract views or representations are necessary if one wishes to capture the design at a higher level and could likewise aid the user of the generator as he constructs a complex design from a number of generated modules. A behavioral model of the generated circuit, for example, could be used in a functional simulator. A network of devices could be used in a switch-level or analog simulator. A schematic diagram could be used for documentation purposes.

Our approach to developing a design generator environment is one that will support such multiple representations. In order to efficiently capture design expertise, the correspondence between such representations will be made evident.

1.1 Design Generator Model

Given the need to support multiple representations, what might an ideal environment for developing generators look like? One such environment is shown in Figure 1. A frontend processes input parameters and performs a variety of manipulations on the input.

The second element of the environment is a database or "model" which guides the generation process. The model fills the gap between the frontend and the generator programs. One element of the model is a list of the instance-specific input parameters provided by the frontend – the "catalog" file. The other two parts of the model contain information about the entire family of circuit designs. The declarative description specifies how the various representations of the circuit are to be assembled. The leaf cells are the representation-specific primitives referenced in the declarative description. In the case of the layout, the leaf cells are simply the mask geometries. In the case of the network representation, the leaf cells may be behavioral models or subnetworks of devices.

The division of information about a circuit family between the declarative description and the leaf cells is arbitrary. Most of the information may reside in a few leaves, as in the case of a network description of a PLA that references two behavioral models – one for the "and" and one for the "or" plane. Another network description for the same family of PLAs may reference all transistors in the design.

Finally there are the circuit generators themselves, which take the information in the model and produce the representations. These programs are independent of the particular circuit family, which is described entirely in the declarative description and the leaf cells. The burden of circuit analysis and construction

should be placed on the generator programs. The idea is to free the designer from as much detail as possible and at the same time keep the notation simple.

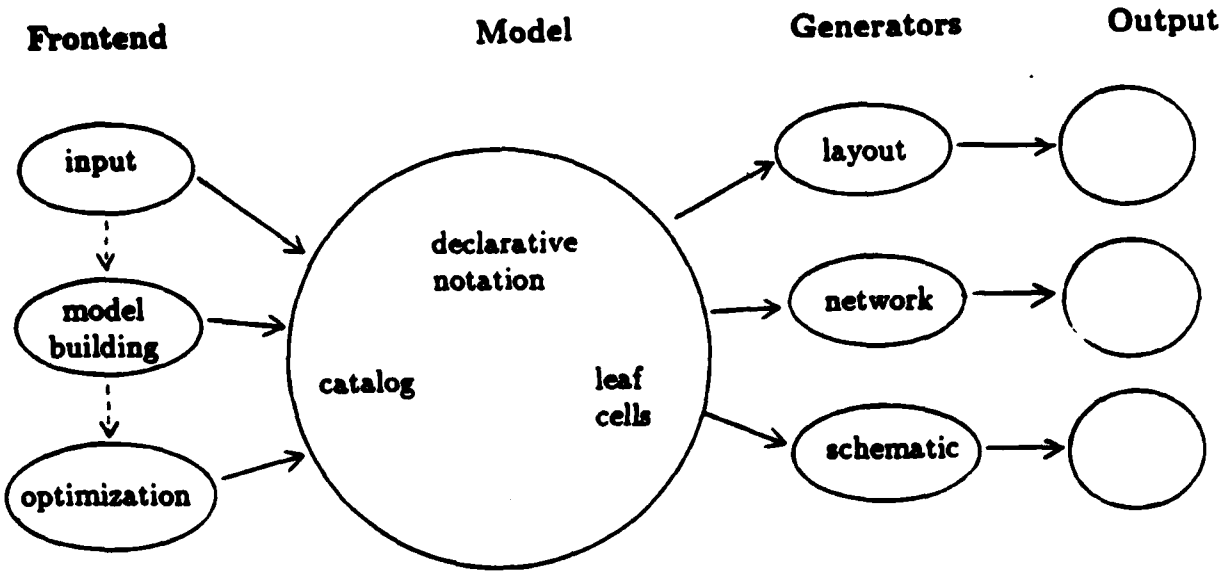


Figure 1: Circuit Design Generation

In looking at Figure 1, it is apparent that we have partitioned the generation process into three separate parts. The advantage gained by doing this is that we can now use the most appropriate language (procedural or declarative) for each part. The frontend is logically written in a procedural language that is best for interfacing to the user, checking input parameters, and performing transformations upon the input. Optimally, the notation describing how circuit families are constructed is declarative, as the objects it describes are static. The generators are representation-specific and may be written in specialized languages.

The key to the scheme we have proposed is the development of a notation for the declarative description. The notation must be expressive enough to describe naturally both geometric structure as well as network information. It must be parametric and sufficiently robust to describe entire families of circuit designs. It must be hierarchical as this is the basic paradigm used in design, and it must be easy for designers to write.

In the remainder of this section we look at languages that describe multiple representations of circuits and outline our own goals. In Section 2, the salient features of the notation will be presented. In Section 3, the generator programs will be described in the context of an example.

1.2 The Multiple Representation Problem

The declarative description is an example of a hardware description language (HDL). HDL's describe to varying degrees functional behavior, network topology and geometrical structure.

Among numerous examples of HDL's is *ADLIB* [Hill 79], a Pascal-based language for specifying circuit behavior. When combined with a topological description, the resulting assembly of behavioral models may be simulated in the *SABLE* environment. The VHSIC Hardware Description Language (VHDL 87) is a language for describing both network topology and circuit behavior. This language does not, however, support semantics for the generation of layout.

There have been numerous efforts to develop languages that describe geometrical structure. The Regular Structure Generator (*RSG*) [BAMJI 85] employs the concept of inherited interfaces to build hierarchical specifications of circuit layouts. *RSG* forms the basis for a layout generator system. Escher [Clark 85] is a layout specification system that allows recursively defined circuits.

There are a few HDL's that allow specification of all three elements - functional behavior, network topology, and geometrical structure. In the functional programming language μFP [Sheeran 83] the behavior specification implies a floorplan and routing. Because of the algebraic properties of μFP, transformations can be made that retain identical functional behavior and allow floorplans and routing to be modified. an important limitation of the μFP behavioral specification is that it doesn't allow explicit representation of state.

Another effort to represent all three types of information is *Zeus* [Lieberherr 83]. *Zeus* is a strongly typed procedural language that allows specification of both signal behavior as well as floorplanning information.

1.3 Goals in the Development of the Notation

The notation for the declarative description has four major goals:

1. Network Topology and Geometric Structure
   The notation must allow a hierarchical specification of both network topology and geometrical structure in a way that emphasizes the correlation between the two representations. In order to make this correlation more apparent we avoided explicit representation of behavioral information, leaving it to be specified in leaf cells. This is in sharp contrast to such languages as *Zeus*.

2. Representation of Entire Circuit Families
   As part of the environment in Figure 1, the notation must allow sufficient parametrization to describe entire circuit families. At the minimum this requires loops and conditionals.

3. Simplicity and Naturalness
   It is important that designers be able to capture circuit designs compactly and expressively. A declarative notation is the obvious choice.

4. Technology independence
   One of the major problems with many powerful layout languages is the technology dependence that can creep in. Although elements of technology can be introduced into the notation (e.g. in a network of MOS devices), a mechanism should be present for hiding such details in the leaf cells In this way the circuit features common across technologies can be emphasized.

2 Declarative Descriptions

2.1 Overview

The declarative description consists of two parts: (1) a declaration, which includes the name of the circuit, the type of the representation, a list of parameters, the names of leaf cells, and a set of imported functions; and (2) a collection of statements used to describe an entire circuit family. A description can be regarded as a set of objects (leaf cells or abstract objects) and a set of relations among these objects.

The syntax of this high level description is designed to be close to that of the "C" programming language. The Extended Backus Naur Formalism (EBNF) definition for the declarative description can be found in [Liem 86].

2.2 Declaration

The syntax of the declaration is of the form:

```
NAME <circuit_name>;

TYPE <representation_type>;

PARAMETER <parameter_list>;

LEAF CELLS <cell_list>;

FUNC <function_list>;
```

<representation_type> is either LAYOUT, SCHEMATIC, or NETWORK. <parameter_list> is a list of inputs to the description. These values are obtained from the catalog file and serve to make the description instance-specific. <cell_list> names all the leaf cells that are used in the description. The leaf cell is the lowest level object in the hierarchy of a description. <function_list> is a list of the (optional) functions that aid the circuit description. For example, 'binary' is a function which returns the value of a specific bit in the binary representation of a number.

2.3 Objects

Leaf cells are the primitive objects on which the operators are applied and out of which abstract objects are built. For example, a leaf cell can be the drawing of a NAND gate used for the schematic description of a decoder or it can be the physical layout of a half-adder used for the layout description of a multiplier. A leaf cell can be instantiated as many times as desired by specifying the number of repetitions. For example, if "lcell" is a leaf cell, then "(| (lcell(n)))" is an object which is a collection of n copies of "lcell"; the relations among these copies will be defined by the operator "|".

Abstract objects are created to provide designers with the mechanism to describe a circuit representation hierarchically so that most of the details at one level of the hierarchy are truly hidden from all higher levels. An abstract object can be defined recursively. An alias of a leaf cell, an array of leaf cells or a

composition of heterogeneous leaf cells are each abstract objects. Moreover, an array of abstract objects, a group of heterogeneous abstract objects or a composition of these two are also abstract objects.

Abstract objects can be instantiated as many times as desired by providing the appropriate arguments. Thus, "(--(row[i](i=0..4)))" is an object which is a collection of five objects whose relations are defined by the operator "--". Abstract objects are considered global. Thus, object names within a description must be unique, *i.e.* "row[i]" *in one abstract object refers to the same object as* "row[i]" in another, although, depending on the value of i, these may be quite different.

Given the features of leaf cells and abstract objects mentioned above, each description can use many levels of abstraction. The highest level of the hierarchy is a single abstract object – the circuit that the designer intends to describe. At the lowest level, the circuit is a collection of leaf cells. The description of a representation of a circuit is recursive in nature; each abstract object is specified as a collection of lower level objects. Since an abstract object may be defined after it is used, the description of an object promotes "top-down" design or "stepwise refinement".

2.4 Operators

The operators which are used in our declarative descriptions can be arranged in the following groups: (1) connection, (2) arithmetic, (3) relational, (4) logical, and (5) assignment ("=").

The connection operators take objects as arguments and produce objects as results. The first four operators ("--", "--^", "|", and "|^") are used to combine objects into more complicated abstract objects.

For NETWORK descriptions, these operators are all identical and cause creation of an object made up of the argument objects connected according to the signal lists given with each object. For example, "A[in, out] = B[in, mid] | C[mid, out]" means that "A" is made of "B" and "C", with the first signal of "A" being the first signal of "B", the second signal of "A" being the second signal of "C", and the second signal of "B" being connected to the first signal of "C".

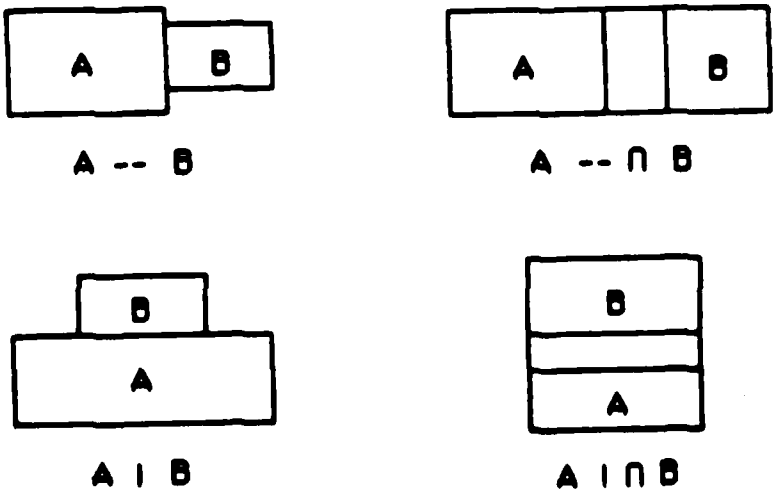


Figure 2: Geometric operators for combining objects

For LAYOUT and SCHEMATIC descriptions, these operators cause creation of an object made up of the argument objects positioned geometrically. These positions are shown in Figure 2. The amount of overlap for operators "--^" and "|^" is controlled by attributes of the leaf cells.

The remaining connection operators, mirror and rotate, have no meaning in NETWORK descriptions.

Arithmetic, relational and logical operators are used as in conventional programming languages. The arithmetic operators are '+', '-', '*', '/', '**' (exponentiation), and '%' (modulus). The relational operators are "<", "<=", "==", ">=", ">", and "!=" (not equal to). There are two types of logical operators; the logical connectives include "&&" (and), and "||" (or), while the bitwise logical operators include "&" (bitwise and), "|" (bitwise inclusive or), "^" (bitwise exclusive or), and "~" (one's complement).

2.5 Control Constructs

Two fundamental control constructs are provided to enhance the expressiveness of a description: IF (decision making) and looping. IF is used to specify conditions. Looping is expressed in either of two forms. The first form provides the number of times for repetition; for example, "(--(X(m)))" represents m horizontally joined instances of X. The second form provides the upper bound, lower bound and step of the loop index; for example, "(| (X[i](i=4..0,-2)))" represents 3 instances of X, with X[4] situated at the bottom, X[2] in the middle, and X[0] on the top.

3 Declarative Descriptions for a Decoder

This section provides examples of descriptions for three circuit representations of a family of precharged decoders with n inputs and 2**n outputs. The descriptions given are for the network, the layout and the schematic diagram. Two different descriptions are given for the network representation to illustrate the manner in which the notation facilitates the design approach of step-wise refinement.

3.1 Network Description

A network description specifies the connectivity between a number of hierarchically structured objects. At the lowest level, these objects are the leaf cells referred to by the description. Leaf cells for a network description contain NC source code expressing the behavior of the object represented by the leaf cell. NC, or Network C, is a language similar to C with additional features to specify behavior, time delays and connectivity [Beckett 86].

3.1.1 High level

The first network description of the decoder is a very simple, high-level description (Figure 3).

```
NAME        decoder;                                          (1)
TYPE        NETWORK;                                          (2)
PARAMETER   n;          /* number of inputs */                (3)
LEAF CELLS  dec_row;                                          (4)
INPUT       x, clock;                                         (5)
OUTPUT      y[2**n];                                          (6)

{                                                             (7)

decoder[x,                                                    (8)
        (,(y[i](i=2**n-1..0) )),                              (9)
        clock ]                                               (10)

    =  ( | ( dec_row[row, x, clock, y[row]] (row=2**n-1..0) ) ); (11)

}                                                             (12)
```

Figure 3: High-level network description of a three-input decoder.
(The numbers in parentheses on the right side of a line are for reference purposes only.)

Since this first network description is at a very high level of abstraction, it uses only one leaf cell (line 4) and has two levels of hierarchy. In a sense, only the top of the hierarchy is described - it consists of a set of n**2-1 interconnected dec_row objects (11), which are modeled as matching functions (see Fig. 4).

In addition to the items in the declarative part of the layout and the schematic descriptions, network descriptions also have declarations of i/o signals ((5), (6)) which may be either single signals or vectors

The output signal y is declared implicitly as a vector. A vector is equivalent to a bus and may therefore have one dimension only. The number of bus signals is given in angle brackets. An automatic expansion of a vector takes place if the unindexed vector (as opposed to an indexed vector component) appears in an object definition.

The purpose of an i/o declaration is to declare a signal as an input signal or an output signal. If a signal appears in both an input declaration and an output declaration, the signal is a bi-directional i/o signal.

Sets of signals may be specified using a looping construct with a comma as the loop operator (9). For example, (,(y[i](i=2**n-1..0))),)) is equivalent to the signal list y[2**n-1], ..., y[0].

The object decoder has the signals x, y[2**n-1], ...,y[0] and clock in its parameter list. Note that x is the integer representation of a bus of binary signals. The parameter list of dec_row consists of the variable 'row' and the signals x, clock, y[row]. The value of the variable 'row' depends on the particular instance of dec_row.

The leaf cell dec_row, used in the high-level description of the decoder, is very similar to an ordinary C function (Figure 4). NC functions, like this one, model the behavior of some part of a circuit. The variable type 'network' is a special NC data type through which nodes (signals) are referenced. (Ordinary variables in the notation, such as 'row', take on specific values when an instance of an object is generated. They are also passed to models as network variables.) Network variables are similar to formal parameters in a C function definition. The model, however, is invoked only if a variable declared as 'network trigger' changes.

```
dec_row()
{
    network trigger row_nr, in, clk;
    network         select;

    if (  clk    == 1
        && row_nr == in) {
        select = 0;
    } else {
        select = 1
    };

}
```

Figure 4: The leaf cell dec_row, referenced in the high-level description of the decoder.

The 'network trigger' parameters of the model (row_nr, in, and clk) receive their values from the leaf cell instance in the description (i.e. from row, x, clock). The model uses these values to compute the new value for select, which is bound to the variable y[row].

3.1.2 Low level

The second network description of the decoder (Figure 5) is considerably more detailed than the first one. (As a guide to understanding the description, see the schematic diagram for a decoder with n=3 inputs in Figure 9.) It uses three leaf cells and has four levels of hierarchy. It illustrates in more detail some of the features of the notation for network descriptions discussed in the previous example.

The description represents a network of nMOS and PMOS devices. The network structure has been completely determined and most of the behavior specification is reduced to that of individual devices (except for the leaf cell 'in'). A level of description intermediate between this and the high-level description discussed previously would be the definition of each row as a precharged nand function with appropriate input bits x[col] and x_bar[col].

Note that, with the exception of the leaf cells and the parameter lists coming with the various objects, the structure of the network description discussed here is very similar to the layout and schematic descriptions discussed in the following sections.

```
NAME        decoder;                                                    (1)
TYPE        NETWORK;                                                    (2)
PARAMETER   n;                                                          (3)
LEAF CELLS  in, nfet, pfet;                                             (4)
INPUT       x[n], clock;                                                (5)
OUTPUT      y[2**n];                                                    (6)
VECTOR      x_bar[n], node[n];                                          (7)
FUNC        binary;                                                     (8)
{                                                                       (9)

decoder[(,(x[i](i=n-1..0) )),                                           (10)
        (,(y[i](i=2**n-1..0) )),                                        (11)
        clock ]                                                         (12)
        =   in_row[x, x_bar]                                            (13)
          | (|(dec_row[row, x, x_bar, clock, y[row]](row=2**n-1..0)));  (14)

in_row[x, x_bar] = (|(in[x[col], x_bar[col]](col=n-1..0)));             (15)

dec_row[row, x, x_bar, clock, sel_line]                                 (16)
      =   nfet[clock, GND, eval_node]                                   (17)
        | select[row, 0, x[0], x_bar[0], node[0], sel_line]             (18)
        | (|(select[row, col, x[col], x_bar[col], node[col],node[col-1]] (19)
                                                    (col=n-2..1)))      (20)
        | select[row, n-1, x[n-1], x_bar[n-1], eval_node, node[n-2]]    (21)
        | pfet[clock, Vdd, sel_line];                                   (22)

select[row, col, in, in_bar, source, drain]                             (23)
        = nfet[    in, source, drain], IF binary(col+1, row) == 1       (24)
        = nfet[in_bar, source, drain];                                  (25)

}
```

Figure 5: Low-level network description of the decoder.

In addition to the implicit vector declarations in the i/o declarations, this description has explicit VECTOR declarations for signals that are internal to the circuit. The expansion of these vectors works just as the expansion of an i/o vector described previously.

The scope of signal names is local to an object definition. Signals within an object definition are connected by name. For example, the sel_line of out_proc[clock, Vdd, sel_line] on line (22) is connected to the sel_line of the first select on line (18).

3.2 Layout Description

The layout representation in Figure 6 directs the construction of a decoder whose floorplan is shown in Figure 7. The leaf cells of the description are Magic files [Ousterhout 85]. The layout generator produces a hierarchy of Magic cells with the decoder being the root cell. These cells may be used like any other Magic cells, i.e. they may be viewed or edited with Magic or may be incorporated into a larger design.

In this example all of the cells tile neatly. The notation is, however, sufficiently flexible to acomodate offset cells as well as overlapping ones. For this purpose, the alignment operators `--^` and `|^` are used in combination with labels present in the leaf cells. With these operators, cells are aligned so that the labels with the same name are superimposed.

```
NAME            decoder;
TYPE            LAYOUT;
PARAMETER       n;
LEAF CELLS      route_l,  in, route_r, eval, out, one, zero;
FUNC            binary;

{

   decoder      =    in_row
                   | ( | (dec_row[row] (row = 2**n-1..0)));

   in_row       =    route_l
                  -- (--(in(n)))
                  -- route_r;

   dec_row[row] =    eval
                  -- (--(select[row,col] (col=n..1)))
                  -- out;

   select[row,col]
                = one,          IF binary(row,col) == 1
                = zero;
}
```

Figure 6: Layout description of the decoder.

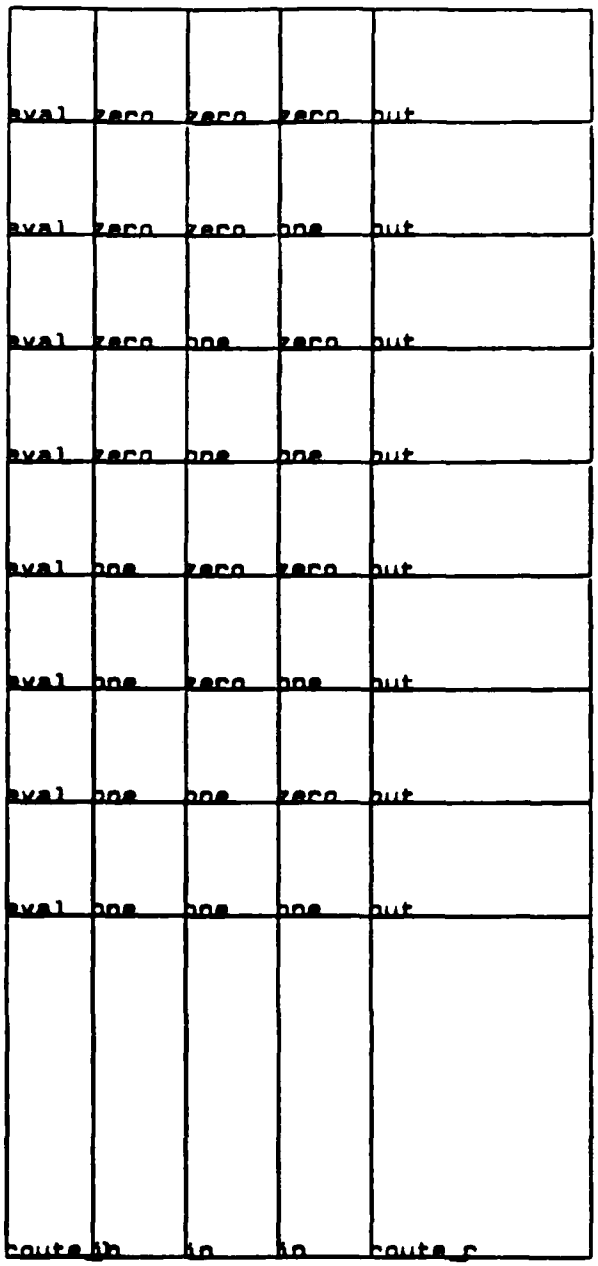


Figure 7: Floor plan of the layout of a three input decoder produced by the layout generator.

3.3 Schematic Description

The schematic representation in Figure 8 directs the construction of a PostScript [1] [Adobe 85] file for a schematic diagram of the decoder. Schematic output is always a picture and is intended for documentation purposes only. The schematic leaf cells are files containing PostScript source code which draw the appropriate symbols or wires. These leaf cells are combined into a single file with appropriate "placement" information. A facility exists for labelling portions of the diagram by string substitution when the leaf cell is referenced. As an example, when "charge_out" is specified in "dec_row[0]", "clock_label[row]" evaluates to the string "clock", which is placed in the output PostScript file.

Figure 9 shows the schematic diagram of the decoder produced by the schematic generator from the schematic of Fig. 8.

```
NAME            decoder;
TYPE            SCHEMATIC;
PARAMETER       n;
LEAF CELLS      in, charge_out, eval, one, zero, blank;
FUNC            binary, strcat, int2str;
{

decoder =    in_row
           | (|(dec_row[row] (row=2**n-1..0)));

in_row  =     blank
           -- (--(in[strcat("sel",row_label[col])] (col=n-1..0)))
           -- blank;

dec_row[row]
        =    eval[clock_label[row]]
          -- (--(select[row, col] (col=n..1)))
          -- charge_out[clock_label[row], strcat("out", row_label[row])];

select[row,col]
        = one, IF binary(row,col) == 1
        = zero;



/* labels */
row_label[row]= int2str(row),                                IF row < 10
        = strcat(int2str(row/10), int2str(row%10)), IF row >= 10 && row < 100
        = "**";

clock_label[row] = "clock", IF row == 0
                 = "";

}
```

Figure 8: Schematic description of the decoder

[1] PostScript is a trademark of Adobe Systems Incorporated

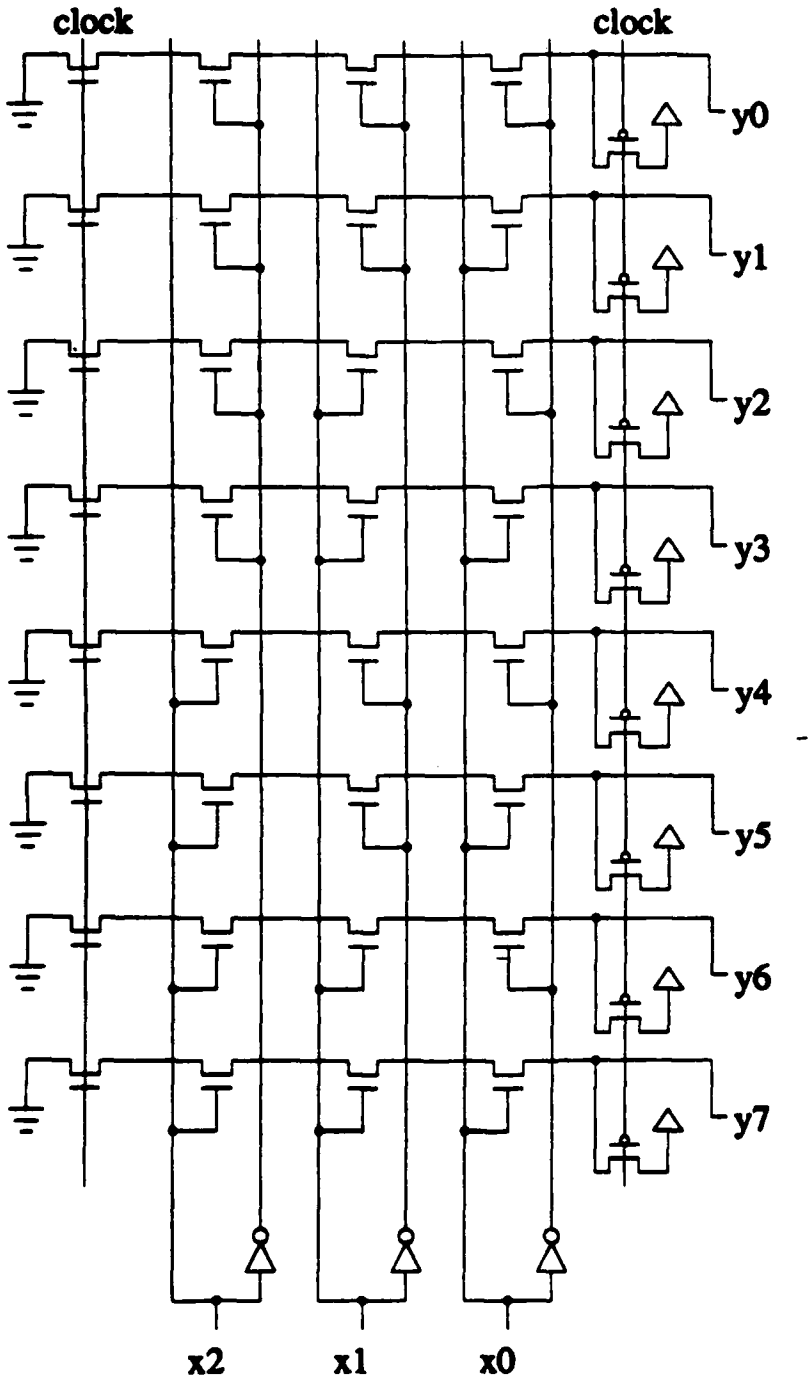


Figure 9: Schematic diagram of a 3 input decoder produced by the schematic generator from the description of Fig. 8

4 Conclusion and Future Work

The high and low level network examples combined with the layout illustrate the application of the notation to top-down design of circuit families. We also note that the three low level descriptions - network layout and schematic diagram - share a common hierarchy; the shared objects provide a natural link between the three circuit views.

We have applied the notation to a variety of decoders as well as a family of NxM Baugh-Wooley multipliers. It has demonstrated utility both as a means of capturing information about circuits at a variety of levels, and as an intermediate database in a design generator environment.

Several improvements could be made that would enhance its usefulness. One is to allow arrays of parameters so that encoded circuits such as ROM's and PLA's could be expressed succinctly.

Another improvement is the use of signal names in the layout description. By including such information, layout generation could be freed from the constraint of requiring interlocking leaf cells. By adding more analysis of cell borders to the layout generator, cell extensions as well as routing could be employed to interface cells according to the specified signal connections.

5 Acknowledgements

We gratefully acknowledge the support of DARPA for this work under contract #MDA903-85-K-0072.

References

[Adobe 85] Adobe Systems Inc., "Postscript Language Reference Manual", (1985).

[Bamji 85] C. Bamji, C. Hauck, and J. Allen, "A Design by Example Regular Structure Generator", *Proc. of the 22nd Design Automation Conference*, pp 16-22, (June 1985).

[Beckett 86] W. Beckett, "MOS Circuit Models in Network C", *Proc. of the 23rd Design Automation Conference*, pp 171-8, (June 1986).

[Beckett 85] W. Beckett, "Coordinate Free LAP", Technical Report #86-07-01, Department of Computer Science, University of Washington, (July 1986).

[Clarke 85] E. Clarke and Y. Feng, "Escher – A Geometrical Layout System for Recursively Defined Circuits", Research Report CMU-CS-85-150, Dept. of Computer Science, Carnegie Mellon University, (July 1985).

[Hill 79] D. Hill, "ADLIB-SABLE User's Guide", Computer Systems Lab., Technical Report No. 177, Stanford University, (1979).

[Lieberherr 83] K. Lieberherr and S. Knudsen, "*Zeus*: A Hardware Description Language for VLSI", *Proc. of the 20th Design Automation Conference*, pp 17-23 (June 1983).

[Liem 86] M. Liem, "Declarative Descriptions for VLSI Generators", Masters Thesis, University of Washington, Dept. of Computer Science, Technical Report 86-09-03, (June 1986).

[Mayo 83] R. Mayo and J. Ousterhout, "Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool", Proc. of the 20th Design Automation Conference, pp 270-6, (June 1983).

[Ousterhout 86] J. Ousterhout, W. Scott, R. Mayo and G. Hamachi *editors*, "1986 VLSI Tools", EECS Dept., University of California at Berkeley, (1986).

[Sheeran 83] M. Sheeran, "μFP - An Algebraic VLSI Design Language", Ph.D. Dissertation, Oxford University (November 1983).

[VHDL 87] VHDL Language Reference Manual, IEEE 1987.

APPENDIX B

Apex: Two VLSI Designs for Generating Parametric Curves and Surfaces

Tony DeRose, Mary Bailey, Bill Barnard, Robert Cypher,
David Dobrikin, Carl Ebeling, Smaragda Konstantinidou,
Larry McMurchie, Haim Mizrahi, Bill Yost

Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

The interactive design of parametric curves and surfaces places a tremendous computational burden on general-purpose graphics workstations. We describe two architectures for a VLSI co-processor chip that generates a large class of spline descriptions extremely quickly. This architecture is based on a triangular computation that generates points on a curve in a data-flow fashion. The first chip, Apex I, maps this data-flow structure directly into silicon. The second chip, Apex II, performs the same computation in a more flexible way that allows the generation of higher degree curves at the cost of lower performance. This paper briefly reviews the theory underlying the triangle computation, focusing instead on the design and implementation of the Apex chips.

1 Introduction

It is now becoming quite common to design geometric objects with the aid of an interactive computer modeling program. In this type of design, the designer may begin with a mental image of the desired shape, the goal being communication of that shape to the system. The system in turn stores an internal representation that is robust enough to support the display, analysis, and possible manufacture of the object.

For "free-form" shapes such as the outline of a character in a typography system or the body of an automobile, spline representations have become popular. The design of a curve typically begins by having the designer specify a sequence of controlling points, collectively called a *control polygon*. It is the responsibility of the system to transform the control points into a smoothly varying spline curve. Of course, there are many ways the system could perform such a transformation, two possibilities of which are Bézier and Lagrange curves (see, for instance, [Bartels et al '87] and [Boehm et al '84]), as shown in Figure 1. Surfaces can be similarly defined by a network of control points, commonly called a *control net*.

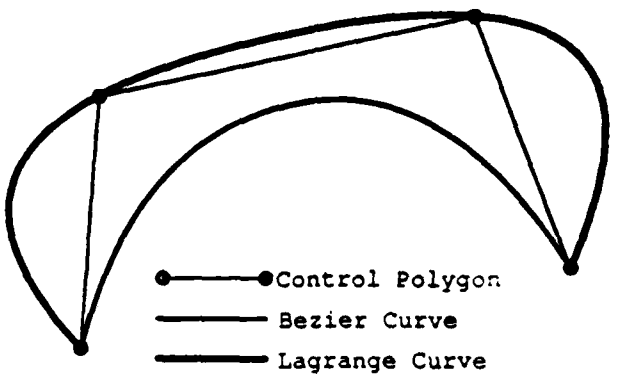


Figure 1: *Bézier and Lagrange Curves*

Based primarily on the visual appearance of the curve, the designer may wish to move a control point, thereby changing the shape of the curve. Ideally, the system would dynamically track the pointing device and redraw the curve in real-time, *i.e.*, at least thirty times per second. Even more desirable is the ability to dynamically deform spline surfaces. Unfortunately, dynamic recomputation at these speeds is simply not possible on conventional serial processors.

In this paper, we describe the design and implementation of two related VLSI architectures that are capable of dynamically recomputing points on line segments, conic sections, and spline curves and surfaces at real-time rates. Unlike previous methods of parametric spline generation such as forward differencing [Lien et al '87] and recursive subdivision [Cheng et al '85] that can only generate a single type of spline description, the architectures we present, called the Apex architectures, are unique in their ability to quickly generate a large class of spline descriptions, a class containing virtually all splines in widespread use [DeRose and Holman '87]. These two single-chip architectures, Apex I and Apex II, are related in that they each perform a special form of geometric calculation called a *triangular computation* (see Section 2). The two architectures differ in that Apex I uses a pipelined multiprocessor to achieve the highest possible performance while Apex II uses a highly optimized serial processor to achieve greater generality in the types of curves it can produce.

Although work on a prototype is still in progress, we envision using an Apex chip (either I or II) as a co-processor device in a graphics workstation. The host initializes the device by configuring it for the curve or surface scheme of interest (Bézier, B-spline, Lagrange, etc.), then supplies the control points for the particular curve or surface to be generated. A short time later points lying on the curve or surface begin

to emerge from the Apex chip. Estimates based on the fabrication of an initial design indicate that a single Apex chip will generate parametric surfaces at the rate of over one million points per second.

The presentation is structured as follows: In Section 2 we briefly motivate and introduce the triangle architecture. In Section 3 we present the overall structure of Apex I and II and discuss the advantages and disadvantages of each. The remaining sections present the details of the two chip designs.

2 Triangular Computations

The Apex architectures are based on the theory of certain discrete probability distributions known as *urn models*. The application of urn models to computer-aided geometric design is currently under development, primarily by Goldman [Goldman '83] and Barry [Barry '87]. Rather than introducing the theory in the context of urn models, in this section we briefly review an equivalent characterization based on ideas that more closely embrace the issues involved in defining and implementing the Apex architectures. For a more detailed discussion of the advantages of the Apex architectures over previously proposed methods for curve and surface generation, see [DeRose and Holman '87].

Geometric design systems typically map the controlling points $\mathbf{V}_i$ into a parametric curve $\mathbf{Q}(t)$ according to the formula

$$\mathbf{Q}(t) = \sum_i \mathbf{V}_i B_i(t), \qquad 0 \le t \le 1$$

where the functions $B_i(t)$ are "weighting", "blending", or "basis" functions, usually polynomials or piecewise polynomials, chosen to endow the curve with a given set of properties.[1] For instance, if the blending functions are chosen to be the Bernstein polynomials

$$B_i^d(t) = \frac{d!}{i!(d-i)!} t^i (1-t)^{d-i}$$

then the curve

$$\mathbf{Q}(t) = \sum_{i=0}^{d} \mathbf{V}_i B_i^d(t), \qquad t \in [0,1]$$

is a Bézier curve of degree d.

Such a curve can be displayed on a graphics screen by computing points on the curve for various values of the parameter t, connecting adjacent points of evaluation with straight lines to obtain a piecewise linear approximation to the true curve. Naturally, the larger the number of points of evaluation, the closer the approximation will be to the true curve.

In the case of Bézier curves, an elegant method, due to de Casteljau [Boehm et al '84], for computing the point on the curve corresponding to a fixed but arbitrary parameter value t can be stated as:

```
B( V_0, ..., V_d, t)
    /* Return Q(t) using de Casteljau's Algorithm */
    for i ← 0 to d do
        V_i^0 ← V_i
    endfor
    for j ← 1 to d do
        for i ← 0 to d − j do
            V_i^j ← (1 − t)V_i^{j−1} + tV_{i+1}^{j−1}
        endfor
    endfor
    return V_0^d
```

[1] For a good discussion of how blending functions influence the resulting curve, the reader is encouraged to consult [Bartels et al '87].

For our purposes, it is convenient to view de Casteljau's algorithm as a data-flow graph, as shown in Figure 2 for the case of a cubic curve. Circles denote addition nodes, and the labels on the arcs of the graph indicate that data flowing along the arc should be multiplied by the value of the label before being input to the incident addition node.

de Casteljau's algorithm for a cubic Bézier curve can therefore be viewed as a three-level labeled digraph with a regular triangular interconnection, henceforth called a *triangular computation*. In general, computation of a point on a Bézier curve of degree d can be viewed as a triangular computation with d levels.

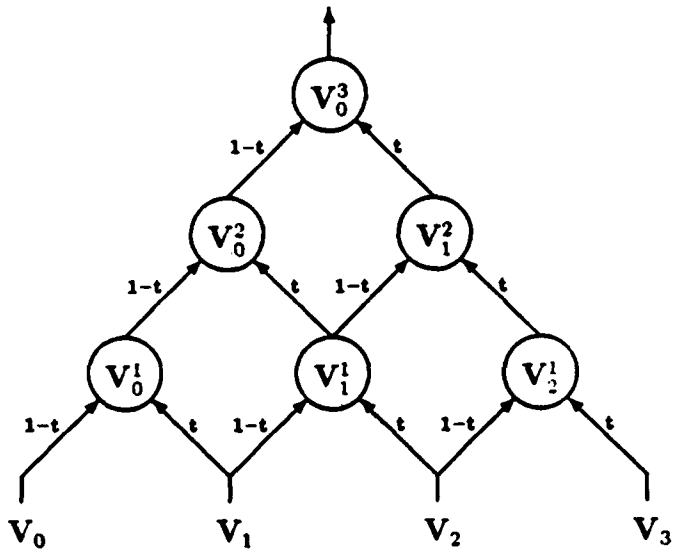


Figure 2: *Cubic Triangular Computation*

Each node V_i^j in a triangular computation has two incident arcs, one from the left and one from the right. For the de Casteljau algorithm, all left arcs are labeled with the function $L(t) = 1-t$ and all right arcs are labeled with the function $R(t) = t$. The theory of urn models states that de Casteljau's algorithm can be generalized to encompass other blending functions, and hence other curve schemes, by extending triangular graphs to allow arbitrary functional labels on the arcs, subject to three restrictions. With the notation that $L_i^j(t)$ and $R_i^j(t)$ denote the left and right labels incident upon the node V_i^j, the restrictions can be stated as:

1. All labels must be linear functions of the parameter t.
2. $L_i^j(t) + R_i^j(t) = 1$ for all i and j.
3. $R_0^d(t) = t$.

For a justification for these restrictions, see [DeRose and Holman '87]. Let us, for the moment, briefly examine how general triangular computations are used to define parametric curves. Just as for de Casteljau's algorithm, a point on a curve generated by a triangular computation with labels

$$L_i^j(t), R_i^j(t), \qquad j = 1, \ldots, d, \quad i = 0, \ldots, d-j,$$

is defined to be the value produced at the apex of the triangle. A particular technique can be described by a triangular computation if it is possible to appropriately choose the functional labels so as to obtain an algorithm for computing points on such a curve. For instance, we have already seen that Bézier curves can be obtained by setting all left labels to $1 - t$, and all right labels to t, thereby resulting in de Casteljau's algorithm. Perhaps less obvious is the ability to generate arbitrary B-splines, cubic Catmull-Rom curves, Pólya curves, and Lagrange curves. Figure 3 demonstrates the labels necessary to generate a uniform cubic B-spline; a detailed assignment of labels for the other curve schemes can be found in [DeRose and Holman '87].

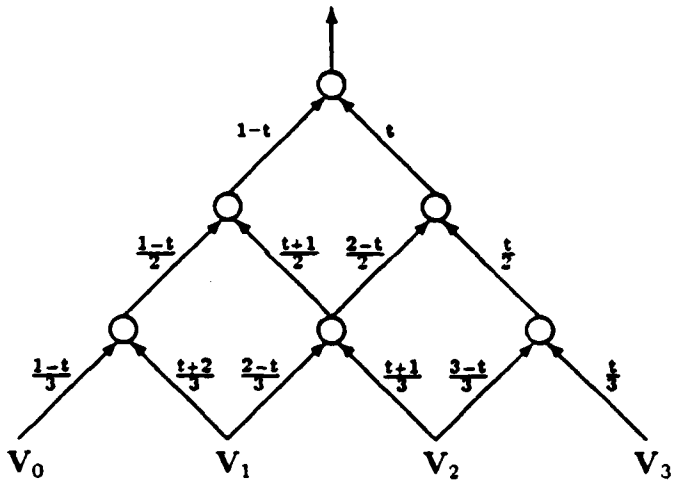


Figure 3: *Uniform Cubic B-spline Triangle*

3 Architectural Issues

The Apex architectures are special purpose machines designed to perform triangular computations. Apex I directly maps the triangular data flow graph into hardware so that a different processing element is associated with each node in the graph. In contrast, Apex II is essentially a serial processor optimized to evaluate triangular computations in a more flexible way to allow the generation of curves of higher degree.

The natural mapping of the structure of the computation directly into silicon in the Apex I design leads to a very high performance pipelined system. Since the control points remain constant while the points along the curve are generated, the host processor can easily supply input data fast enough to keep the pipeline full. Thus the utilization of the chip circuitry approaches 100%.

This performance exacts a price in terms of generality. In Apex I, a triangular array is mapped into the plane which raises the difficult problem of partitioning a grid onto several chips. The bandwidth requirement along the edges of partitions requires that the triangle be contained wholely within a single chip or divided into partitions of limited size. Apex I embeds the entire triangle computation on one chip, limiting the degree of the curves that can be generated to the level of integration possible. Our current design uses 2 micron

technology to generate curves of degree 3, which is sufficient to cover the vast majority of applications. However, if higher degree curves are required, we project that more aggressive circuit design and 1.2 micron technology would allow curves up to degree 5.

The precision with which Apex I can compute values is also limited by the area of the chip. The size of the multiplier grows as b^2 where b is the number of bits of the operands. Apex I limits the values of the labels to the range $[0, 1]$ because the error incurred by the interpolation grows with the size of the label. This restriction means that Apex I cannot generate some types of curves like Lagrange curves. However, most common curves such as Bézier and uniform and non-uniform B-splines are still handled. One way to use Apex I to generate a more general class of polynomial curves is to perform a change of basis. For example, the control points of the Lagrange curve would be transformed to control points of a Bèzier curve and then generated as usual. Although this change of basis is also numerically unstable, it can be done on the host where extended precision is available.

While Apex I takes advantage of the structure of the computation to attain very high performance, it is clear that a more flexible way of performing the triangle computations would be advantageous. Apex II was designed with two goals – to handle larger degree curves and increase the accuracy of the calculations to allow a larger family of curves.

In Apex II a single processor element is fed data from several RAMs, with a ROM controlling the sequencing. Because there is only one processor element, the area required by Apex II is significantly smaller than that for Apex I. This allows use of a larger word size, giving greater dynamic range and hence accommodating a larger family of curve types. For this increase in flexibility, Apex II sacrifices speed: for curves of degree d, throughput is reduced by a factor of $2/(d(d+1))$ compared to Apex I.

The current design of Apex II uses 2 micron technology to generate curves up to degree 7. In addition it employs a large enough word size to allow calculation of uniform Lagrange curves in addition to Bèzier and B-splines.

4 Apex I

Each processor element in the dataflow graph performs the same computation:

$$\mathbf{V}_o \leftarrow L(t)\mathbf{V}_l + R(t)\mathbf{V}_r$$

where $\mathbf{V}_l$ and $\mathbf{V}_r$ are the left and right inputs of the processor. Since $L(t) = (1 - R(t))$ this computation can be simplified to:

$$\mathbf{V}_o \leftarrow \mathbf{V}_l + (\mathbf{V}_r - \mathbf{V}_l)R(t) \tag{1}$$

which requires only one multiplication and two additions.

Since $R(t)$ is a linear function, it can be computed using forward differencing as long as the values of t at which the curve is evaluated are uniformly spaced. Each processor element then comprises a forward difference unit to generate the label $R(t)$ and an interpolation unit as shown in Figure 4.

A further simplification is made for the processors at the base of the triangle. The inputs to these processors are the control points of the curve which remain constant over all values of t. Equation 1 is then a linear function of t and can itself be generated using forward differencing that interpolates the points $\mathbf{V}'_l$ and $\mathbf{V}'_r$ where $\mathbf{V}'_l$ is the output of the processor element for $t = 0$ and $\mathbf{V}'_r$ is the output for $t = 1$. The additional calculation required to compute $\mathbf{V}'_l$ and $\mathbf{V}'_r$ is left to the host since it is done only once for each curve. For the common case of Bézier curves, note that $\mathbf{V}'_l = \mathbf{V}_l$ and $\mathbf{V}'_r = \mathbf{V}_r$.

The points (x, y) on a 2-dimensional curve can be computed by the two independent parametric equations $X(t)$ and $Y(t)$. It is convenient, however, to generate the coordinates in pairs so that points can be displayed immediately. There are two ways to do this with Apex I depending on the speed of the host system. The first is to use two Apex I chips, one computing $X(t)$ and one $Y(t)$, doubling the rate at which curves are drawn. Alternatively, the Apex I chip can be programmed to generate up to three coordinate values per point by multiplexing the coordinates through the pipeline. The processors at the base of the triangle, which

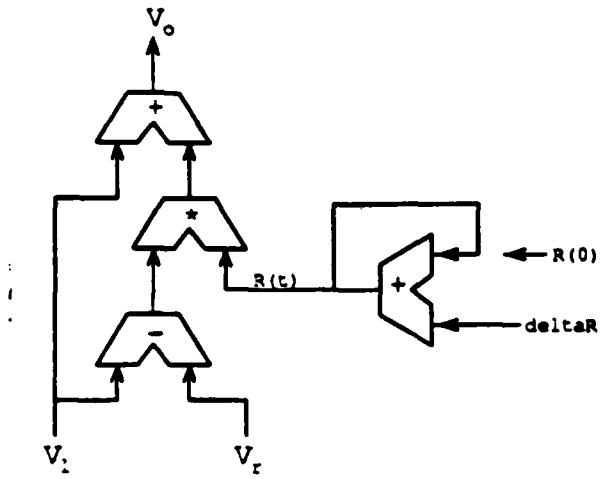


Figure 4: *One processor element.*

we call *vertex generators*, serially generate up to three different coordinate values. The remaining processors remain unchanged except that the same label is used for all coordinates of the same point.

4.1 Generating Surfaces

Triangular computations can also be used to generate surfaces. The control points comprising the control net of the surface are blended together to form a set of curves, called *parameter curves*, each of which is generated by a triangular computation. A triangular computation can then be used to blend together the parameter curves to generate the surface [DeRose and Holman '87].

This process can be implemented by using one Apex chip in conjunction with host computation. The host is used to serially generate one point on each of the parameter curves. These points are then fed to the Apex chip that generates an entire curve lying on the surface. Relatively speaking, this does not place much of a computational burden on the host. For instance, if we assume degree 3 curves, 100 points per parameter curve, and 100 points for each curve lying on the surface, then the host must compute 400 points, while the Apex chip computes 10,000 points.

While Apex I can generate the three coordinate values for curves in 3-space, some sort of post-processing must be done to render the object in 2 dimensions. This includes perspective transformations and z-buffer algorithms which many graphics workstations already provide.

4.2 Vertex and Label Generation

The processors at the base of the triangle and the units that generate the labels both use the forward difference method and thus are very similar in design. The two units differ in that the vertex generator produces up to three coordinate values for each point while the label generator produces only one label per point. The design of the vertex generator is shown in Figure 5.

Forward differencing is performed by initializing the pipeline registers with the initial value of each coordinate. The appropriate delta values are then accumulated using a 32-bit pipelined adder. The high order 16 bits form the vertex values sent to the processors at the next level in the triangle. The low order 16 bits are used to maintain sufficient accuracy and are truncated. The coordinate values are held in a circular pipeline and the delta value of the appropriate coordinate is added at each cycle. The length of this pipeline is set according to the number of coordinates being generated. The adder itself is pipelined in two stages with the low-order 16 bits of a value being incremented on one cycle followed by the high-order 16 bits on the next.

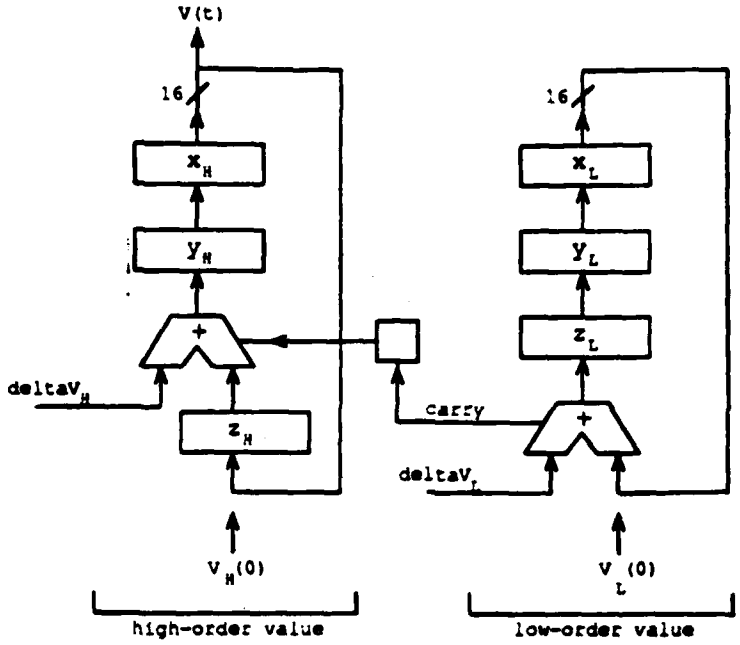


Figure 5: *The vertex generator.*

The label generator is identical to the vertex generator except that there is only one pipeline register and the increment is performed only once per point.

4.3 The Interpolation Unit

Each processor element performs the interpolation of Equation 1 using the label $R(t)$ produced by the label generator. The initial subtraction is a simple modification of the adder while the final addition is folded into the multiplier by treating V_l as an additional partial product. The multiplier is based on the modified Booth multiplier using the sign generate method described in [Annaratone '86]. The multiplier multiplies the 16-bit signed value $V_r - V_l$ by the 15-bit unsigned fraction $R(t)$ producing a 16-bit signed value as the result. Since only the 16 most significant bits of the multiplier result are kept, the low order 16 bits of the partial products are discarded within the multiplier array. However, since these discarded bits can generate a carry into the high order result, an additional carry resolve unit has been added to each row which computes the carry generated by the discarded bits. The final carry is then used as the carry into the final 16-bit adder used to resolve the carries in the high order results from the multiplier array. This 16-bit adder replaces a 32-bit adder without any loss of accuracy.

The 16-bit adder used in the multiplier is a precharged Manchester carry adder with carry bypass. This same adder is used elsewhere to perform forward differencing and the initial subtraction in the interpolation unit. The time to perform one 16-bit addition was chosen as a convenient cycle time for the chip and thus the multiplier is pipelined such that the time through each stage conforms to this basic cycle time. The amount of pipelining and the resulting throughput can be increased at the expense of an increased number of pipeline registers.

4.4 Pipeline control and I/O

Each value flowing through the pipeline has associated with it a control word that indicates the processing that should be performed on that value. This control word controls the multiplexing of the appropriate

coordinates by the vertex generators and controls the generation of the label values at the correct times. It is also used to mark the beginning of a new curve which causes the vertex and label generators to begin using new data values. The data registers specifying the initial and delta values for the forward differencing are double-buffered so that the I/O section can load the data for a new curve while the previous curve is being generated. In this way, breaks in the pipeline are avoided.

The pipeline control is generated by a finite state machine that interacts with the I/O unit via a control register. Upon initialization, the control unit waits for the I/O unit to set the registers in the vertex and label generators with the definition of a curve. The I/O unit then sets a 'Ready' bit in the control register which causes the control unit to generate the appropriate control words for the pipeline. The number of coordinates generated per point is indicated by a field in the control word while the number of points to be generated on the curve is specified by a count register.

The chip has been designed so that a 16-bit value is generated each clock cycle. Since this will be too fast for many systems, a WAIT signal can be asserted to stall the pipeline until a new value can be handled.

4.5 Performance

The multiplier part of the processor element was designed and fabricated through MOSIS. A yield of 15 of 18 chips was achieved with a clock rate of 5MHz. Based on this performance, we expect the final Apex I chip to generate 2-dimensional curves at the rate of 2 million points/sec. and surfaces at 1.5 million points/sec. A final version of Apex I will be sent for fabrication in early Fall '87 with an initial prototype system comprising a Sun 3 graphics workstation with an Apex I co-processor operational in Spring '88.

5 Apex II

In contrast to the parallel implementation of Apex I, Apex II uses a single high performance processing element to perform the triangle computations in the order shown in Figure 6. The advantage of sequencing the computation in this order is that the control can be defined recursively. That is, the computation sequence for a degree $d-1$ curve is a subsequence of that for a degree d curve.

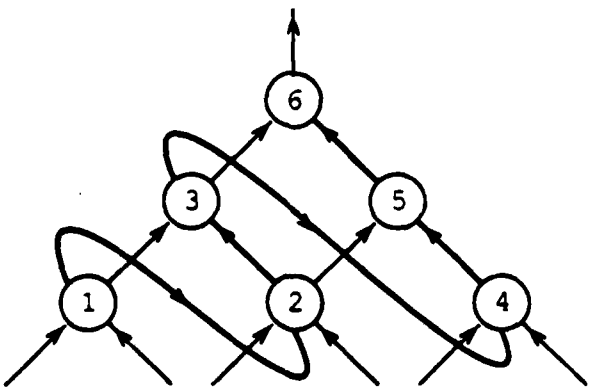


Figure 6: *Sequencing the triangle computation in Apex II.*

As nodes in the triangle are evaluated in sequence, many of the calculations made by the processor element require the output of the previous node as an input (e.g. node 6 requires the output from node 5). However, the processor element is pipelined with P stages and thus the triangle cannot be evaluated in sequence without some null cycles. This problem was solved by introducing a loop over P values of t within the overall loop through nodes in the triangle. This means that after the evaluation is done for the Pth

value of t, the result for the 1st value of t will have progressed through the pipeline and be available as an input for the next node in the sequence.

A temporary storage RAM is used to save node outputs that are not needed until later in the processing sequence. For example, the output of node 3 is used by node 6 and must be stored while nodes 4 and 5 are evaluated. The organization of this memory can be considered to be P pages each with $d-1$ words, where d is the maximum degree that the chip can handle. Each page corresponds to a different value of t. To generate the write address, a pipeline counter cycles through the pages while another counter cycles through the $d-1$ words on a page. The read address is a more complicated function determined by the output of a ROM and the pipeline counter.

Figure 7 shows how this data is handled. **Pipecount** corresponds to the current t value and gives the page address; **pestate** addresses the specific node under evaluation in the triangle; and **nextvl** and **nextvr** are the read word addresses out of the ROM.

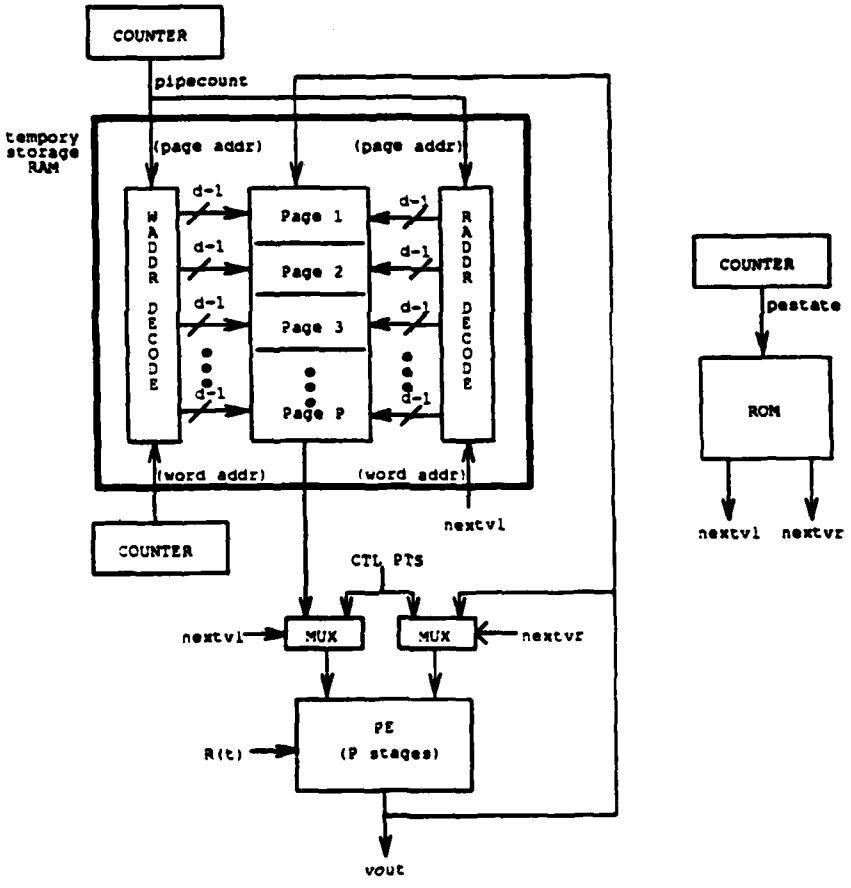


Figure 7: *Pipelining the data for the PE*

5.1 Label Generation

Label inputs to the nodes are of the form $R(t) = mt + b, t \in [0,1]$ where the slope m and intercept b are coefficients that must be provided for each node. The function $mt + b$ is computed directly in the Apex II chip by a processor element that is a subset of the triangle processor. The coefficients m and b are stored

in a RAM and are loaded in the initialization phase of the circuit operation. The address to the RAM is simply the **pestate** and changes only as the node under evaluation is updated.

The t generation circuitry cycles through P values of t for each node in the triangle. Once the top of the triangle is reached a new set of t values replaces the old. The value of t is incremented by a user definable dt value to generate each succeeding value. When the value of t is increased to a value greater than one a signal is generated that indicates that evaluation of the current set of control points is complete.

5.2 Loading and Testing

There are four control inputs that control all functions of the chip:

- The start signal resets all registers and must be asserted before any data is run.
- The run command is asserted to cause the chip to carry out the triangle calculations.
- The load signal activates the loading of data onto the chip. All storage locations for data inputs to the chip are mapped onto the address space of an 8 bit load address bus. When the load signal is asserted data on a bidirectional load/dump bus is stored into the location specified by the load address bus. The control points, label coefficients, degree of the triangle and dt must be loaded before the triangle can function.
- The dump signal is for testing only and enables the storage location specified by the load address bus to be written back onto the load/dump bus. The dump function is nondestructive in that the chip operation can be halted, a dump performed and the chip operation then continued without affecting the output of the chip.

5.3 Implementation

An Apex II chip is currently being designed with the following parameters: The maximum degree, d, is 7, the number of bits, b, is 16 and the number of pipeline stages, P, is 4. The chip is based on a 7.9 x 9.2 mm. die using MOSIS 2-micron scalable CMOS design rules. Each module and the padframe was produced with the procedural layout system *CFL* [NW LIS, '87] and the graphical layout editor *Magic* [Scott et al, '86].

The processor element design is identical to that used in Apex I as described in Section 4.3. The choice of 16 bit accuracy in the processor element is sufficient for Bezier and uniform B-spline curves. An extension to 20 bits would allow generation of 7th degree uniform Lagrange curves. Because the processor element design, as well as the storage arrays, are generated procedurally as a function of word size, this extension is straightforward.

Several memory modules are used to store control points, label coefficients and temporary values. The control points and the label coefficients are stored in static single-port RAMs organized as 8 words by 16 bits and 28 words by 32 bits respectively. Temporary values are stored in a static dual-port RAM organized as 24 words by 16 bits; a dual-port is required in order to fetch the next value needed as an input to the processor element and simultaneously store the current output of the processor element. Control signals are generated by a control ROM which is a 28 word by 14 bit two transistor ROM. These memory modules have been designed to operate with 50ns cycle times.

Several datapath modules are required for processing temporary values as well as generating t values. These bit-sliced modules consist of registers, muxes, tri-states, and latches; assembly and routing of these elements is handled semi-automatically with *CFL*.

Based on the performance of the fabricated multiplier unit described in Section 4.5, we expect to generate degree 3 curves at a rate of 1 million points/sec. This figure assumes the use of two chips, one to compute X values and the other Y values. The design of Apex II will be completed in early Fall '87 and then fabricated through MOSIS.

6 Conclusion

Apex I and II solve the same problem in two different ways to achieve different goals. In this section we discuss how the designs differ and what the tradeoffs are. The parameters of the design are performance. the maximum degree of the curve that can be generated, and the precision with which values are computed. This last factor is important because some types of curves like Lagrange curves (for which the label values exceed 1) can only be generated if the computational error can be controlled.

The two different chip architectures use silicon area for two different goals. The Apex I design uses it to obtain the greatest possible parallelism by integrating an entire triangle but is restricted to curves of relatively small degree. Since Apex II has just one processor element it can use the remaining area in a number of ways. The most important is to increase the degree d of the curves generated. This only increases the amount of temporary storage by a factor of d in contrast to Apex I whose total area depends on d^2. Another way to use the area is to increase the pipelining and thus the throughput of the processor element. This again increases the amount of temporary storage by a factor of P. Finally, the area can be used to increase the precision of the computations thus allowing a larger class of curves to be generated. Thus Apex II achieves a much greater degree of flexibility and generality compared to the relatively inflexible architecture of Apex I. However, where the curves are of modest degree and the highest performance is required, Apex I is the design of choice.

The two chip architectures presented in this paper cover a wide range of possible implementations with very high performance and low flexibility at one end and high flexibility with somewhat lower performance at the other. With the addition of a single co-processor Apex chip, a graphics workstation can incorporate a very high performance and general curve drawing capability.

7 References

[Annaratone '86] Marco Annaratone, *Digital CMOS Circuit Design*, Kluwer Academic Publishers, Norwell, Mass. (1986).

[Barry '87] Phillip J. Barry, *Urn Models, Recursive Schemes, and Computer Aided Geometric Design*, Ph.D. Thesis, Department of Mathematics, University of Utah, Salt Lake City, Utah (June, 1987).

[Bartels et al '87] Richard H. Bartels, John C. Beatty, and Brian A. Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, Morgan-Kaufmann Publishers, Inc., Los Altos. California (to appear).

[Boehm et al '84] Wolfgang Boehm, Gerald Farin, and Jürgen Kahmann, "A Survey of Curve and Surface Methods in CAGD," *Computer Aided Geometric Design*, Vol. 1, No. 1, July, 1984, pp. 1–60.

[Cheng et al '85] Fuhua Cheng, Kuen-Rong Hsieh, Rei-Ron Huang, and Yeh-Hao Chin, "Bézier Curve Generator: A Hardware Approach to Curve Generation," Proceedings of the *Second International Symposium of VLSI Technology, Systems, and Applications*, May 1985, Taipei, Taiwan, pp. 278–281.

[DeRose and Holman '87] Tony D. DeRose and Thomas J. Holman, "The Triangle: A Multiprocessor Architecture for Fast Curve and Surface Generation," Technical Report #87-08-07, Department of Computer Science, University of Washington, FR-35, Seattle, WA. 98195, August 1987.

[Goldman '83] Ronald N. Goldman, "An Urnful of Blending Functions," *IEEE Computer Graphics and Applications*, Vol. 3, No. 7, July 1983, pp. 49–54.

[Lien et al '87] Sheue-Ling Lien, Michael Shantz, and Vaughan Pratt, "Adaptive Forward Differencing for Rendering Curves and Surfaces," *Computer Graphics*, Vol. 21, No. 4, July 1987, pp. 111–118.

[NW LIS '87] Northwest Laboratory for Integrated Systems, *VLSI design Tools Reference Manual, Release 3.1*, Department of Computer Science, University of Washington, Seattle, Washington (February 1987).

[Scott et al '86] Walter S. Scott, Robert N. Mayo, Gordon Hamachi and John K. Ousterhout, editors, *1986 VLSI Tools*, EECS Dept., University of California at Berkeley, Berkeley, California (1986).